USAISEC

US Army Information Systems Engineering Command
Fort Huachuca, AZ 85613-5300

U.S. ARMY INSTITUTE FOR RESEARCH
IN MANAGEMENT INFORMATION,
COMMUNICATIONS, AND COMPUTER SCIENCES

ACE: LESSONS LEARNED
Vol. 1

September 30, 1992

ASQB-GC-92-023

AIRMICS
115 O'Keefe Building
Georgia Institute of Technology
Atlanta, GA 30332-0800

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188
Exp. Date: Jun 30, 1986

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICAION AUTHORITY N/A	3. DISTRIBUTION/AVAILIBILTY OF REPORT N/A
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ASQB-GC-92-023	5. MONITORING ORGANIZATION REPORT NUMBER(S) N/A
6a. NAME OF PERFORMING ORGANIZATION AIRMICS 6b. OFFICE SYMBOL *(If applicable)* ASGB-GCN	7a. NAME OF MONITORING ORGANIZATION N/A
6c. ADDRESS (City, State, and Zip Code) 115 O'Keefe Building Georgia Institute of Technology Atlanta, GA 30332-0800	7b. ADDRESS *(City, State, and ZIP Code)* N/A
8b. NAME OF FUNDING/SPONSORING ORGANIZATION AIRMICS 8b. OFFICE SYMBOL *(If applicable)* ASGB-GCN	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* 115 O'Keefe Bldg. Georgia Institute of Technology Atlanta, GA 30332-0800	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. DY10 / TASK NO. 07-03 / WORK UNIT ACCESSION NO.
11. TITLE *(Include Security Classification)* ACE: LESSONS LEARNED Vol. 1 — UNCLASSIFIED	
12. PERSONAL AUTHOR(S) Gerald McCoyd, Adrienne Raglin	
13a. TYPE OF REPORT 13b. TIME COVERED FROM_____ TO_____	14. DATE OF REPORT *(Year, Month, Day)* September 30, 1992 15. PAGE COUNT
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUBGROUP	ACE, Rapid Prototyping, Rapid Development, ETIP

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This report describes AIRMICS involvement with the development of three prototypes using AT&T's Application Connectivity Engineering (ACE) Tool. The prototyping tool, ETIP, and the communications and file transfer tool, ESCORT, were used. This report includes the lessons learned during this project. The principle lesson learned was that success in using ACE demands an environment that provides the programmers with a wide array of tools to simplify and coordinate their work. In this report summaries of the experiences we had with ACE are included, as well as, presenting the tools that we developed and those which were supplied to us by other organizations, most notable Information Foundation. This report is intended primarily for programmers who are familiar with ACE.

20. DISTRIBUTIONN/AVAILIBILTY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Adrienne J. Raglin	22b. TELEPHONE *(Include Area Code)* (404) 894-3136 22c. OFFICE SYMBOL ASGB-GCN

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

This research was performed under contract DAKF11-91-C-0081 for the Army Institute for Research in Management Information, Communications, and Computer Sciences (AIRMICS), the RDT&E organization of the Army's Information Systems Engineering Command (ISEC). This report is not to be construed as an official Army position, unless so designated by other authorized documents. Material included herein is approved for public release, distribution unlimited. Not protected by copyright laws.

THIS REPORT HAS BEEN REVIEWED AND IS APPROVED

s/
John Gowens, Chief
Communications Network
and Systems Division

s/
John R. Mitchell
Director
AIRMICS

Executive Summary

This report describes AIRMICS involvement with the development of three prototypes: Central Issue Facility (CIF), TRADOC Resource Manager's Information and Decision System-Test (TRMIDS), and Publications Support System (PUBSS) using ATT&T's Application Connectivity Engineering (ACE) Tool. The ACE prototyping tool, ETIP, and the communications and file transfer tool, ESCORT were used. The principal lesson we have learned from this experience is that success in using ACE demands an environment that provides to the programmers a wide array of tools to simplify and coordinate their work. In this report we are summarizing our experience with ACE, presenting those tools which we have developed and those which have been supplied to us by other organizations, most notably Information Foundation.

ACE: LESSONS LEARNED
Vol. 1

U.S. Army Institute for Research in Management Information, Communications, and Computer Sciences

Georgia Institute of Technology

Atlanta, Georgia 30332-0800

Acknowledgment

The following persons have contributed to this report: Gerard McCoyd (AIRMICS), Adrienne Raglin (AIRMICS), Reginald Hobbs (AIRMICS), Cyndy Whitman (SDC-A), Ardie Dempsy (SDC-A), Vaniesa Davis (MBRI), Paul Ofori (MBRI), Sirin Wangspa (MBRI), Byron Jeff (GT), Kit Kamper (GT), Murali Shanker (GT), Joachim Ungruh (GT), Angela Teachey (AT&T).

Introduction

Purpose of This Report

For the past two years, AIRMICS has been involved in the development of three prototypes using AT&T's Application Connectivity Engineering (ACE) Tool. We have used the prototyping tool, ETIP, and the communications and file transfer tool, ESCORT. The principal lesson we have learned from this experience is that success in using ACE demands an environment that provides to the programmers a wide array of tools to simplify and coordinate their work. In this report we are summarizing our experience with ACE, presenting those tools which we have developed and those which have been supplied to us by other organizations, most notably Information Foundation. This report is intended primarily for programmers who are familiar with ACE.

AT&T's Application Connectivity Engineering Tool

The AT&T ACE (Application Connectivity Engineering) technology has been developed as a rapid prototyping tool. The ACE tool consists of ETIP and ESCORT software which run on a AT&T 3B2/600 (or higher).

The AT&T ETIP Designer[1] is a software tool that provides application programmers a fast way to develop user interfaces. ETI (Extended Terminal Interface) is a set of screen management library routines. The AT&T ETI-Prototype Designer, which is referred to as ETIP produces code that makes calls to the ETIP library routines and supports the development of experimental user interfaces for an application. ETIP creates objects which can be: edited without exiting ETIP, connected together to form executable prototypes, and executed to test the branching, appearance, and overall design of the application. ETIP Shell and C objects provide the capability of calling executable programs from either the shell or compiled C programs.

ETIP consists of three toolkits: user interface (UI) toolkit, the ETIP host connectivity toolkit, and the ETIP relational database toolkit. The user interface toolkit is for creation of the user interfaces for the prototype under construction. This is a mandatory toolkit. This provides a fairly quick way for the programmer to create screens, menus, forms, etc., and to link these screens with one another and with processing objects, like C programs, Ada programs, and SQL scripts. The other toolkits are

1. ETIP Designer is a Trademark of AT&T.

optional, but are needed if the application will have host connectivity or relational databases. The relational database toolkit is for application developers to generate software with embedded Standard Query Language (SQL) statements for data manipulation in a relational database manager. All data manipulation statements are permissible, such as storage, retrieval, and modification of tables.

The ETIP host connectivity toolkit is for application programmers to generate software that supports interactive communication between a local distributed system and a remote host system. The AT&T ESCORT/3270 is the application host connectivity tool for ACE. Escort is a programming interface with English like commands based on the UNIX operating system that simplifies and automates access to the host. ESCORT which runs in conjunction with AT&T SNA/3270 or BSC/3270, allows the programmer to access up to four synchronous host sessions, four asynchronous host sessions and two local sessions. ESCORT programming interface provides an effective approach for automated multi-host connectivity. ESCORT applications can check and verify the data entered, automatically update the mainframe applications, review batch output, verify its completion, and generate progress reports. Using ESCORT one can develop simple user screens, evaluate prototype interfaces, perform local editing, send error messages, and capture data from any host application.

Our Experience

There are three applications on which we used ACE: the TRADOC Resource Manager's Information and Decision System - Test (TRMIDS-T), the Publication Support System (PUBSS), and the Clothing and Issue Facility (CIF).

TRMIDS-T purpose is to collect, coordinate and analyze resource data from many databases throughout the Army. The application used shell scripts for the processing objects, and made heavy use of ESCORT. There were no database interactions.

PUBSS is the support for Army's publications management and distribution system. All processing objects were C programs except for interaction with an INFORMIX database which was accomplished with SQL objects.

CIF is the system for maintaining records of equipment issued to individual soldiers. It was written originally by Software Development Center Atlanta in INFORMIX/4GL, and we assisted them in converting it to ACE. All processing objects, including database interactions, were implemented in C programs.

Lessons Learned

During the course of developing these systems, we identified many characteristics of the ACE environment which required different ways of working or the development of software tools. This document covers eight areas where tools were developed to add to the capacity of ETIP.

Prototype Decomposition

In the early stages of development, we discovered that only one user could effectively access a given prototype for developmental purposes at any given time. Unfortunately, this constraint limited the time and productivity advantage available through team development.

To take advantage of team development, we decomposed the application into several ETIP prototypes along functional lines, and created an integrating tool to integrate them at the end of the development; these separate prototypes were then integrated into the final system.

Configuration Control

Even when we decomposed the application, we still needed more than one programmer working on an ETIP prototype so it became necessary to establish a configuration control mechanism to preclude simultaneous attempts at updating the same prototype.

For this purpose, we developed the concept of clean files from which the entire prototype could be reproduced. Used in conjunction with lock files, which restricted effective access to the clean files to only one programmer at a time, these clean files gave us complete control over changes to the prototypes no matter how many programmers were involved.

Non-graphical Display of Logic

The programmer must maintain control of the flow of the prototype during development. The display logic function of ETIP can be used to itemize and display the line structured flow of the objects in the prototype. This function details which screens are associated with what menu items, then outlines the flow of the objects created within the structure. Although a useful tool, this function was not designed for the average prototype developer since it is complicated to follow if the application is large. Additionally, it cannot be used for any type of presentation or documentation purpose. As the system grows in size and becomes more

integrated and complicated, it becomes necessary to maintain a written control of the flow of the system, not only for reference purposes but also as a communication tool with the functional users.

Using information in the display logic, flowcharts were created by using a drawing tool on the Apple Macintosh IIx computer. It would be helpful to have available on ETIP a more readable flowcharting capability with graphical interface.

Data Inconsistency

During the development of any prototype, it is necessary to either have access to the data to be used before development begins or to have accurate documentation of the lengths of the fields. It is important when developing a prototype system that involves data collection and storage that the field lengths outlined match the maximum length of the actual data to avoid data errors and to give a realistic representation of production data. If the fields represented do not match the lengths outlined in the documentation or are not greater than the length of the data, it can cause a misrepresentation of the data to occur at run time.

In order to eliminate the inconsistency, the data fields should be extended to the length of the longest record to avoid truncation of data. This may involve making some minor preparatory corrections to the data fields or, in some cases, redesigning the data storage files or tables. Making changes to either the data files or the tables is easier if a realistic report of the maximum data lengths is available at the beginning of the development.

Difficulty of Editing Forms

Once a form has been created and linked to a data table, it becomes increasingly difficult to make even minor corrections to field attributes. Once a correction is made, each remaining field in the form has to be reexamined for subsequent changes that may have occurred. All fields may not have their attributes altered, but each field must be checked individually for changes, a task which is both time consuming and ineffective.

One way to handle this problem is to make sure during the specification and design phase of the project that the forms that involve data entry are created exactly as they should be with the correct data specifications. However, this frustrates one of the advantages of rapid prototypes, that we can get user feedback after we have shown the user the consequences of the current design. To preserve this advantage, we developed a method for associating attributes with field forms in files external to the ETIP database.

A procedure run at make time then assigns these attributes after ETIP has generated the C code for the form.

Limitation of Shell Scripts

Shell scripts, although useful, do not provide all of the features needed to design a more advanced system. Basic UNIX shell scripts can be used to insert, update or review data in a prototype. But as the request demands more of the data manipulation, or if the designed prototype should ever be used for more than just a prototype, a more advanced method of programming is needed. Shell scripts in some cases do not allow for all of the necessary functionality needed, especially if a system is to be used in a production environment. In addition, shells may not adequately address programming system security and multi-user access.

To obtain the needed functionality in the prototypes, C programs were used to develop programming tasks beyond the capability offered by shell techniques.

Need For Robust ESCORT Scripts

ESCORT scripts enable programs to retrieve data from interactive applications with no human intervention. In order to build an ESCORT script you must have a model of the behavior of the interactive application. Generally, the interactive application will not be under the control of the organization which is running the ESCORT script, in fact that organization may not even be on the mailing list for changes. However, the behavior of an interactive application is not always completely specified in the documentation and if it is used frequently there tends to be slow but continuous modification. Whenever the model built into the ESCORT script becomes erroneous, either because it is incomplete or because the application has been changed, the ESCORT script is in danger of crashing or wasting money and time before the problem is detected.

We ran into a lot of problems like this and we found the only effective way to overcome them is to require the script to verify at each step that it has reached one of the interactive screens that it knows about. Whenever it detects an unknown screen, it should put a copy of the screen in a log file and terminate.

End-user Participation

A useful way to ensure fewer system design related concerns is to solicit the input and participation of the functional end-users throughout the design phase. Even when the end-users have designed or assisted in designing the specification, there is an awakening process that takes place when the product is actually

viewed on the screen. The earlier within the development this joint review begins, the less the time involved later in making corrections or additions to the system. Editing a screen when it has only been formatted is much easier than changing it after it has been integrated with shell scripts and other objects.

The development team should set up periodic check points for design review. At each check point, the functional users should be given the opportunity to evaluate how well the prototype meets their functional and aesthetic requirements.

The following sections will discuss the different procedures and tools that were developed to address the limitations we discovered.

Configuration Management

In any development environment, there is a level of subdivision of the code at which two programmers cannot simultaneously make changes to the code. In a C environment this would occur at the C source module level, in Ada at the package level. In ETIP this occurs at the level of the prototype. The ETIP Designer manages a prototype as an integrated unit which can be accessed by one call to the Designer. For each prototype, there are two directories used by the ETIP Designer. One of these is informally called the working directory. It must be the current directory when the designer is invoked with the name of the prototype. The working directory contains all the source code generated by the designer for this prototype and it contains the subdirectory <prototype-name>:UT. The :UT directory contains all the information which the designer maintains about the prototype.

In order that more than one programer can work on an application, it is desirable to decompose the application into prototypes along functional lines. To maintain control over the prototypes, we have created a set of procedures that must be used by programmers.

The configuration management environment consists of a set of directories which will be referred to by environmental variables all of which are defined in an ENVIRON file. The environmental variables are described in the Appendix.

Many of these directories contain a subdirectory for each prototype, e.g. for an application with two prototypes, X and Y, the directory NEWPROTO would contain subdirectories X and Y. We will not always refer explicitly to the following subdirectories, although the procedures will require their existence.

EDIT contains the master copy of every static file used in CHOICES submenus. See the description of the Submenu Tool.

HELP and FMTHELP contain the text files for each help screen in the prototype. See the description of the Help Tool.

CLEAN contains the current baseline of the prototype.

LOCK implements a locking mechanism so that two programmers cannot simultaneously be attempting to modify a prototype.

NEWCOPY contains the new version of a prototype after it has been changed and tested by the programmer, but before it has been returned to CLEAN.

FORMS contains a description of every field in every form of the prototype including prompts, associated choices files, and edit

criteria. See the description of the Form Tool.

REASON contains the reason that a prototype was taken out of the CLEAN area for modification.

REPORTS contains the report generator code for all reports generated by the prototype.

TESTNEWCOPY provides a place to test the version of the prototype as it exists in NEWCOPY.

INTEGRATE provides a place where all the prototypes can be integrated into the application.

INCLUDE contains application specific .h files.

There may also be an application specific environment. This would be defined in the file APPL/ENVIRON which will be invoked by /usr/ACE/ENVIRON.

The core of these directories is the clean directory; the other directories support this directory. The following section discusse the files maintained in the clean directory.

Clean Files

We maintain the baseline of the application in ASCII files, not in ETIP databases. This gives us both a flexible way to control the configuration and an easy way to transport the development environment.

The CLEAN files contain, in ASCII format, the most recent version of a prototype and a record of past versions. CLEAN contains for each prototype, say X, the following files:

X.exp - a complete unloading of the ETIP data bases with delimiter |

X.saf - a complete unloading of the ETIP data bases with delimiter ,

X.src - a shar file of all the files in the :UT directory that are not part of the ETIP data base

X.txt - a shar file of all the .txt, and H files in the :UT directory. These files are also included in X.src.

X.fil - a shar file of the files listed in the working directory file BUNDLEFILES. This list should contain all the files in the working directory which are needed for the application, but are not generated by ETIP. That is the .c, .o, .sh, .txt files do not belong on this list; they are taken care of by the .exp, .txt, and .src files. BUNDLEFILES should contain the names of awk scripts, lists of constants, and other programmer generated files which reside in the working directory and which are referenced. It should also contain BUNDLEFILES.

For each prototype, there is also a set of files X.exp.o, X.saf.o, X.src.o, X.txt.o, and X.fil.o which contain a cumulative copy of past versions of X. Each time the procedure newclean is executed, the file X.exp is appended to X.exp.o and the file NEWCOPY/X.exp becomes the new X.exp, and similarly for X.saf, etc. A line containing the string "NEW VERSION" and the date separate successive versions of the prototype in all the .o files. In addition, the .exp.o file also contains the reason for which the new version was created.

Description of Associated Code

The following are the procedures that are called to interact with the clean files.

appl.set.up

This must be used to initialize an ETIP application.

In developing an application, we must allow several programmers to access and modify the files. However, each ETIP application should have a control user, i.e. a user with sole authority to modify the sensitive directories like CLEAN, EDIT. The login name of this control user should be stored in the file /usr/ACE/CONTROL which can be accessed by CONTROL.

To set up an ETIP application, the application administrator or the control user must:

1. Create a base directory for the application. This should be accessible for reading and writing by everyone who will work on the application.

2. Create .profile in the base directory with the control user as the owner and sole writer.

3. Have the ETIP administrator enter the login name of the control user into CONTROL. This will authorize the control user to make changes to the application's baseline, i.e. update the CLEAN areas.

4. Enter the following lines into .profile in the base directory:

```
APPL=X; export APPL
MYENVIRON=Z; export MYENVIRON
. /usr/ACE/ENVIRON
```

X is the full path of the application's base directory. Z is the name of a special environment; right now, pubss and embed are the only special environments.

5. Enter . .profile, or log in again, to set the application environment

6. Enter appl.set.up which will create all the required directories.

appl.dump

appl.dump makes a copy only of the essential files of an application and not the total application. It takes two parameters: the home directory of the application, and the path of the directory where the application is to be stored. It copies the following from the home directory of the application: .profile, CLEAN, EDIT, HELP, FORMS, REPORTS, INCLUDE.

appl.restore

appl.restore is the inverse of appl.dump. It restores an application. It is not necessary to run appl.set.up before running appl.restore.

modify a programmer's .profile

Before working on an application, the programmer must insert the following lines into his/her .profile file:

```
APPL=X; export APPL
MYENVIRON=Z; export MYENVIRON
. /usr/ACE/ENVIRON
```

X is the full path of the application's base directory. Z is the name of a special environment; right now, pubs and embed are the only special environments. Then whenever the programmer logs on, he/she will be ready to work on the application.

newproto

A programmer wishing to work on a prototype, say X, moves into the work area, NEWPROTO/X which we shall refer to as the working directory, and issues the command "newproto X". If the file LOCK/X exists, the programmer will be told that someone else is working on the prototype. If the file LOCK/X does not exist, it will be created and it will contain the identity of both the programmer and the working directory. The programmer will be prompted to enter the reason for opening the prototype. This will be entered in the file REASON/X, and the prototype X will be built up in the working directory

from the files in CLEAN. All the .edt files in EDIT will be linked into the working directory. newproto can be issued only in the directory NEWPROTO/X.

bundle

After the programmer has finished modifying and testing the prototype in NEWPROTO, the command "bundle X" should be issued. This command will create the files X.exp, X.saf, X.fil, X.src, and X.txt and store them in NEWCOPY. It will NOT touch the file LOCK/X.

newclean

The command "newclean X" must be issued to return the new version of the prototype to the CLEAN directory. This can only be done by CONTROL. This will take the current copy of the X files in CLEAN and append them to the history files X.exp.o, etc which are also maintained in CLEAN. The words NEW VERSION and the current date and time will be appended to each of the .o files, and the reason for modifying the prototype will be appended to the X.exp.c file. The files in NEWCOPY will be copied into CLEAN, they will be removed from NEWCOPY, the directory X:UT will be removed from the working directory, and the X file will be removed from LOCK.

rmproto

If at any time after the newproto and before the newclean, the programmer issues the command "rmproto X", the X:UT directory in the working directory, the X files in NEWCOPY, and LOCK/X will be removed. No change will be made to CLEAN.

testproto

At any time, anyone can issue the command testproto X to create a copy of the current CLEAN version of X. This is a scratch copy, since any changes made to it cannot be saved in CLEAN. The command testproto can be issued anywhere except in NEWCOPY.

testnewcopy

testnewcopy X will create a copy of the version of X which is in NEWCOPY. This is useful for testing the changes made to X after it was bundled into NEWCOPY. Again, any changes made to this copy cannot be saved in

CLEAN. The command testnewcopy can be issued only in TESTNEWCOPY/X.

stripclean

stripclean can be used to trim the .o files in CLEAN by removing the older versions. This can only be executed by CONTROL.

The Integration Tool

Decomposing an application into separate prototypes along functional lines is useful and perhaps essential for development. However, these prototypes must be integrated in order to produce a complete application. We know of three ways to accomplish this:

1. Integrate the prototypes at the ETIP level, i.e. merge the separate ETIP :UT directories into a single :UT directory. We have not tried this method, but we think that Information Foundation has some experience with it.

2. Integrate the prototypes into a single load module by linking the object modules from all the prototypes. We used this approach on the PUBSS system. See Single Load Module below.

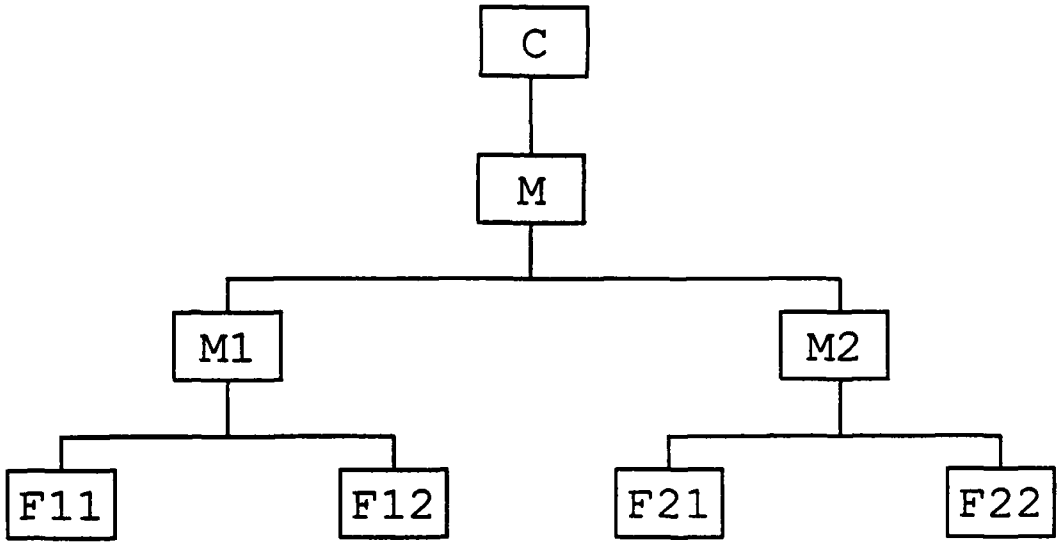
3. Integrate the prototypes by means of a front-end prototype which implements a top-level tree structure for the application, and branches to the appropriate load module depending on the user's selections in this tree. We used this approach with the CIF system. It is fairly straightforward except that the screen must be refreshed whenever the user causes control transfers between load modules.

Single Load Module

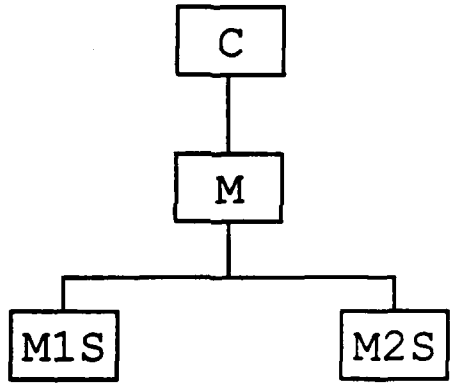
In order to combine several independently developed prototypes into a single load module, it is necessary to create a make file for the entire application which is built up from the makefiles for the separate prototypes. The integration tool accomplishes this task. The following simple example will illustrate the integration tool.

The application FINAL, illustrated on the next page, consists of an introductory C object, C, whose purpose is to open up the databases, a main menu, M, and two subordinate menus, M1 and M2, each of which is the root of a tree. If there is no interaction between the objects in these trees, then we can decompose the development into three prototypes: Sub_top, Sub_1 and Sub_2.

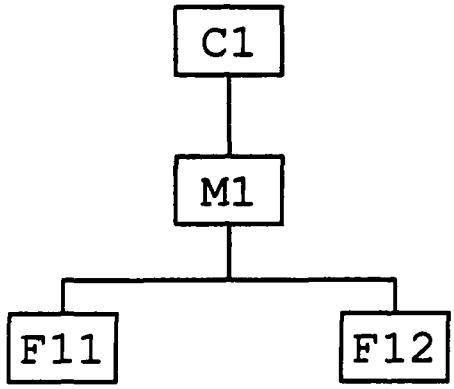
In addition to the objects, like M1, F11, and F12, which are part of the application, each of the prototypes must have additional objects to make them complete prototypes. Sub_1 and Sub_2 must each have an initial object to open its databases and do other initialization; and Sub_top must have stubs M1S and M2S to provide bridges to Sub_1 and Sub_2. The integration tool requires that these stubs be C objects, but their variable names must be the same as the variable names of the objects whose place they are holding.



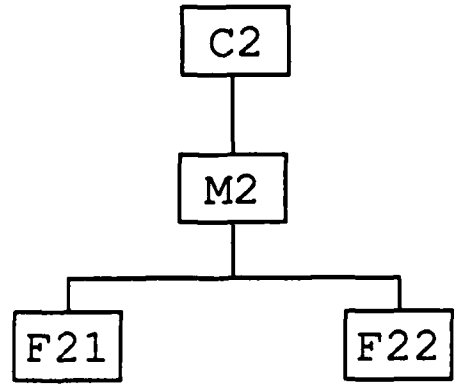
Final



Sub_top



Sub_1



Sub_2

To integrate Sub_top, Sub_1 and Sub_2 into Final, we must eliminate M1S, M2S, C1 and C2, and link M with M1 and M2. The links from M to M1 and M2 have been established implicitly by giving M1S and M2S the same variable names as M1 and M2. Elimination of M1S, M2S, C1 and C2 is accomplished by the integration tool under the direction of the file integrate.data.

The format of the contents of integrate.data is as follows:

a) Comment lines are added by putting a # in the first column.

b) All other lines must be in the following form:

<directory name> [<list of objects to be left out of final prototype>]

where the directory name is the path to the directory where the prototype was developed, and the optional object list is a space separated list of object files not to be included in the final prototype. This is primarily to eliminate any unnecessary setup objects (especially SQL database opens) in the prototypes.

c) The first directory listed must contain the main program.

To simplify integration, the tool automatically drops two types of files:

1) The main.o from all sub-prototypes except the one from the first directory listed.

2) All C objects from the main directory. However, if a C file is put in the optional drop list of the main directory, it will be INCLUDED in the integrated prototype.

For this example the integrate.data file looks like:

```
../sub_top C.o
../sub_1 C1.o
../sub_2 C2.o
```

With this guidance, the integration tool will leave C in the prototype, and eliminate C1 and C2. Following its default rules, it will also eliminate M1S and M2S since they are C objects.

The Submenu Tool

Whenever the valid data for a form field consists of a discrete set of values, a choices submenu attached to the field provides the simplest way for the user to enter the desired value, and for the programmer to validate the data. The field attributes F_REQUIRE and F_MONLY guarantee that only values in the submenu set will be accepted.

There are, however, several problems with using submenus within the ETIP Designer. Because ETIP requires unique labels as well as unique variable names for all objects, two choices submenus defined using the ETIP Designer must have different labels even if they contain the same data. This forces the programmer to maintain a list of alternate labels for the same submenu. The programmer must also type in the prompt for the submenu, and since these prompts are usually of standard form, this leads to the problem of consistency among submenus.

The submenu tool simplifies the use of submenus by providing the following capabilities:

1. Maintain the texts of static choices submenus external to the ETIP :UT directory. This has the advantage of concentrating the maintenance of these files in one place.
2. Insure that all static choices submenus have the same prompt.
3. Permit a submenu to have the same label wherever it appears. The submenu tool assigns the label by modifying the generated code, so there is no longer a requirement for uniqueness.

Procedures

To create and maintain submenu files:

All submenu files for the application are maintained in $EDIT. Each file bears the extension .edt. In the same directory, there is a file called labelmap which associates a label with each edit file. A sample labelmap file follows:

```
chkcode.edt|CHECK CODES
class.edt|CLASSIFICATION
component.edt|COMPONENTS
dastatc.edt|DA STATUS CODES
ddislev.edt|DA DISTRIBUTION LEVELS
gslevel.edt|GS LEVELS
idistlev.edt|DISTRIBUTION LEVELS
```

```
location.edt|LOCATIONS
locstatc.edt|LOCAL STATUS CODE
maintlev.edt|MAINTENANCE LEVELS
mdislev.edt|MACOM DISTRIBUTION LEVELS
par.edt|PAR CODE
states.edt|STATES
subacctc.edt|SUB ACCOUNT CODES
subacctc.lcl|SUB ACCOUNTS
untissf.edt|UNIT OF ISSUE FOR FORMS
untissp.edt|UNIT OF ISSUE FOR PUBS
userpermit.edt|USER PERMISSIONS
yesno.edt|YES/NO
```

To attach a submenu file to a form field

In ETIP, choose sub_menu for a field and enter the edit file name, say states.edt, under File. Nothing else is required. The submenu label and the prompt will be automatically generated during the make process by the procedure modsubslcl.sh.

Description of Associated Code

The following procedure is called to interact with the edit files.

modsubslcl.sh

modsubslcl.sh modifies the generated C code adding the standard prompt and the labels defined in labelmap. It should be called as part of the make process.

The Form Tool

With the ETIP Designer it is easy to create and modify the appearance of a form. However, it is awkward and tedious to maintain the properties of the fields in the form. For example, the addition of a new field to a form erases the characteristics of all fields which follow the new field.

The form tool provides a mechanism for maintaining the field attributes of forms in files that are external to the ETIP database. This provides two advantages:

1. It is easier to modify the data in the form tool files using a text editor with search and replace capabilities, like vi, than it is to step thru the ETIP editor screens to make the same modifications.

2. The addition of new fields to a form will no longer erase the characteristics of other fields that have already been defined but that follow the new field in the left to right, top to bottom sequence.

Using the form tool, you can specify the prompt, pathname of the choices menu, the field attribute flags, and the regular expression for all fields in a form. The only property of a field that cannot be assigned by the form tool is the field branch.

These are the static properties of the field, i.e. the properties assigned to the field at compile time. The form tool assigns these properties by modifying the generated C code for the form; it does not change the ETIP data base.

Form Descriptor Files

The form tool stores the field properties in form descriptor files which are used in the make process to modify the generated C code.

A form_descriptor file has the following structure:

```
Object name = "Add/Change/Delete User"
Variable name = "F_ACDU"
Field: 1 (Row,Col): (0,1)
Label: "Login Name: "
Width: 10
FIELDFLAGS: 0 | F_CLEARIT
PROMPT: ""
REGEX: ""
```

```
EDITS: ""
Field: 2 (Row,Col): (0,2)
Label: "Permissions: "
Width: 20 FIELDFLAGS: 0 | F_CLEARIT | F_MONLY
PROMPT: ""
REGEX: ""
EDITS: "userpermit.edt"
```

There will be a field entry for each field in the form. The field number as in Field: 1, is the sequence number of the field in a left to right, top to bottom ordering.

The row refers to the actual row in which the field appears, but the column is a macro column, i.e. the first, second, etc. field in the row.

Label is the field caption.

Fieldflags are the field attributes.

Prompt is the prompt.

Regex is the regular expression that will be applied to the data. If a regular expression is specified, than F_REGEX must also appear as a fieldflag.

Edits is the name of the file, in $EDIT, to be used as the choices submenu. If there is an edits file, then usually the fieldflag F_MONLY will be present.

The lines Field, Label, and Width are generated from the ETIP database, as will be explained below. They should not be changed or else there will be an inconsistency between the form descriptor files and the ETIP data base.

The remaining lines specify attributes of the field which may be modified within the form tool.

There are three form_descriptor files for each form. They reside in that subdirectory of $FORMS, which has the same name as the prototype. To illustrate the purpose of the three files, suppose we have a form with variable name X and that the form has two fields. Then we will have three files, curr_X, mrg_X, mst_X, and they might look like:

curr_X

```
Object name = "Enter State and Zip"
Variable name = "X"
Field: 1 (Row,Col): (1,1)
```

```
Label: "State:"
Width: 2
FIELDFLAGS: 0 | F_USHIFT | F_MONLY | F_REQUIRE | F_CLEARIT
PROMPT: "Enter the Unit State: Use CHOICES"
REGEX: ""
EDITS: "states.edt"

Field: 2 (Row,Col): (1,2)
Label: "Zip:"
Width: 10
FIELDFLAGS: 0 | F_REQUIRE | F_CLEARIT
PROMPT: "Enter the Unit Zip-code: nnnnn-nnnn"
REGEX: ""
EDITS: ""
```

---

mst_X

```
Object name = "Enter State and Zip"
Variable name = "X"
Field: 1 (Row,Col): (1,1)
Label: "State:"
Width: 2
FIELDFLAGS: 0 | F_USHIFT | F_MONLY | F_REQUIRE | F_CLEARIT
PROMPT: "Enter the Unit State: Use CHOICES"
REGEX: ""
EDITS: "states.edt"

Field: 2 (Row,Col): (1,2)
Label: "Zip:"
Width: 10
FIELDFLAGS: 0 | F_REQUIRE | F_CLEARIT | F_REGEX
PROMPT: "Enter the Unit Zip-code: nnnnn-nnnn"
REGEX: "[0-9]{5}-[0-9]{4}"
EDITS: ""
```

---

mrq_X

```
Object name = "Enter State and Zip"
```

```
Variable name = "X"
Field: 1 (Row,Col): (1,1)
Label: "State:"
Width: 2
FIELDFLAGS: 0 | F_USHIFT | F_MONLY | F_REQUIRE | F_CLEARIT
PROMPT: "Enter the Unit State: Use CHOICES"
REGEX: ""
EDITS: "states.edt"

Field: 2 (Row,Col): (1,2)
Label: "Zip:"
Width: 10
FIELDFLAGS: 0 | F_REQUIRE | F_CLEARIT | F_REGEX
PROMPT: "Enter the Unit Zip-code: nnnnn-nnnn"
REGEX: "[0-9]{5}-[0-9]{4}"
EDITS: "
```

---

curr_X is produced by formfields.sh and splitcurr.sh. It contains the description of the fields from the ETIP database. Note that the flag F_REGEX is not set and there is no data in REGEX, because there is no way to assign regular expressions within ETIP.

mst_X is essentially a copy of curr_X that has been modified to indicate how we want the form to behave. It is used to modify the generated C source code at make time. This is the only way to get regular expressions into the application.

mrg_X is produced by mrgformfields.sh which merges the data from curr_X and mst_X. This merger is needed to insure consistency between the ETIP database and the mst_X file which will be used to modify the generated C code.

The merge process copies curr_X into mrg_X, and then overwrites FIELDFLAGS, PROMPT, and REGEX with the values in mst_X. It reports an error if Field, (Row,Col), or Label differ in curr_X and mst_X, because this indicates that modifications have been made to the form within ETIP. Before mst_X can be used to modify the generated C code, manual changes to mst_X will be required to restore consistency between it and the ETIP database.

The following procedure should be followed to generate form descriptor files for an application:

If there are no mst_X files,

a. run formfields.sh to create currformfields.

b. run splitcurr.sh to create the cur_X files.

c. copy currformfields to mstformfields.

d. run splitforms.sh to create the mst_X files.

e. modify the mst_X files to the desired values of field attributes, prompts, and regular expressions.

f. run validmst.sh to validate the correctness of the mst_X files.

If there are mst_X files,

a. run formfields.sh to create currformfields.

b. run splitcurr.sh to create the cur_X files.

c. run mrgfrmfld.sh to generate the mrg_X files and the mrg.ERR files.

d. working on the mrg_X files, correct any inconsistencies revealed between the cur_X and mst_X files.

e. copy the mrg_X files to mst_X files, using mrgtomst.sh.

f. modify the mst_X files to the desired values of field attributes, prompts, and regular expressions.

g. run validmst.sh to validate the correctness of the mst_X files.

The form descriptor files, mst_X, produced by this process will be used by the make process, testmake or lclmake, to modify the generated C code.

Remarks

In its present form, the form tool does not modify the submenu labels or prompts, this is done by the submenu tool. These tools should be combined.

Description of Associated Code

The following procedures are called to interact with the files that manipulate and update the forms.

formfields.sh

formfields.sh creates the currformfields file for an application. This is a preliminary step in creating the cur_X files. The currformfields file contains a form description for each form in the application, i.e. a list of the attributes of the fields in all forms of the application. The file is created in the $FORMS directory for the application. If there is a prototype in the current directory whose name matches $1, it is used as the source; else the version in $CLEAN is used.

modformslcl.sh

modformslcl.sh changes the .c files for the forms in the current directory of the application. It is run as part of the make process.

mrgfrmfld.sh

mrgfrmfld.sh merges the data in each cur_X file with the data in the corresponding mst_X file to create a mrg_X file for each form in a prototype, where X is the variable name of the form. It reports in mrg.ERR the following error conditions:

NOMST - there is a cur_X form but no corresponding mst_X form

NEWCURR - cur_X and mst_X differ in names and/or locations for one or more fields.

mrgttomst.sh

mrgtomst.sh replaces each mst file for an application with the corresponding mrg file.

splitcurr.sh

splitcurr.sh decomposes a currformfields file into cur_X files, one for each form in the currformfields file. splitcurr.sh removes all files of the form cur_* in $FORMS/$1. Then it reads the current forms file, currformfields, and creates a file named cur_name for each form in the application, where name is the variable name for the form, e.g. F_ACDUI.

splitforms.sh

splitforms.sh is used only once for each application. It is used to create the initial set of mst_X files. Before running splitforms.sh, the cuurformfields file should be copied into a new file called mstformfields. Then splitforms.sh decomposes the mstformfields file into mst_X files, one for each form in the mstformfields file.

validmst.sh

validmst.sh checks the mst_X files for valid data, such as prompt lengths.

The Help Tool

The help tool provides the following capabilities:

1. Maintain help screen texts external to the ETIP :UT directory. This has the advantage of permitting persons who are not familiar with ETIP to maintain the help screen text.

2. Ensure that all help objects return control to the object from which they were invoked, and do so through the return branch. This frees the ETIP programmer from one easily overlooked task.

3. Concentrate in one procedure, currently an awk script, the specifications for help screen appearance, such as the prompts at the bottom of each frame.

4. Enable a single help screen to be used in more than one place in an application with each help object that uses it still returning control to the object from which it was called.

Procedures

Create and maintain help screen text

All help screen text for a prototype, say X, will be maintained in a subdirectory of $HELP. The subdirectory will have the name of the prototype, i.e. $HELP/X. The files in $HELP/X will be called help files.

A help file should have the same name as the variable name of the ETIP object with which it will be associated. Since menu items do not have variable names, help files associated with them should be given the variable name of the menu with a subscript equal to the sequence number of the item within the menu.

There are two potential problems associated with these virtual variable names for menu items:

The first occurs when an additional item is inserted in a menu. This is a real problem and it will require renaming the help files for all items following the insertion.

The second problem concerns the limit of 10 characters to a menu variable name. By appending one or two characters to the menu name to create the virtual menu item name, we could exceed this limit. This limit was imposed because ETIP appends .txt to the variable name when creating the help file text. Since we never create a file with the extension .txt we never exceed the limit on file name lengths.

To assist in assigning names to these files, the procedure

call "mnufrmlst.sh X" will create in the local directory a list of all forms, menus, and menu items in the prototype X along with their variable names, internal id's, labels, and the internal id's of associated help objects.

Here is a sample mnufrmlst file generated for a small prototype.

List of Menus and Forms for sysadmin in /usr2/isms/pubscntl/NEWCOPY

	Var Name	ID	Label	Help ID
F	"F_ACDU"	1:5	"Add/Change/Delete User"	1:29
F	"F_ACDUB"	1:16	"Add/Change/Delete User Browse"	1:30
M	"M_SAM"	1:27	"System Administration Menu"	1:31
	"M_SAM1"	0:16	"1. Add/Change/Delete User Menu"	1:32

Notice that the forms are preceded by the letter F and the menus by the letter M, and the menu items have been given fictitious variable names. The names in the Var Name column must be given to the help files. In other words, if you want to attach a help screen to the menu object M_SAM, you create a file called M_SAM in directory $HELP/X and insert the help screen text into this file.

The final column shows, by printing the ID number, whether or not there is a help object associated with the form or menu.

When creating a help text file, center the title on the top line (no margins) and type in up to 72 characters per line of text. Do not include margins or navigation directions or prompts (i.e., "Press RETURN to continue" at the bottom.) The help tool will use these help files to generate the code needed for ETIP and to insert the standard margins and prompts.

If the same text is to be used in two or more help objects, simply link the help files together. There will be multiple formatted help files and help objects but they will all contain the same text, and the text needs to be maintained in only one place.

Create formatted help files

The procedure call "formathelp.sh X" will make a copy in $FMTHELP/X of each help file in $HELP/X. This formatted copy will be properly spaced for display in an ETIP frame, and it will contain the appropriate prompts, such as "Press RETURN to continue.". The formatted files will bear the same names as the help files except that the first character of the name will be H rather than F or M.

The formatted files in this area should never be modified except thru the formathelp.sh procedure.

Create help objects in the prototype

If the prototype X is in $NEWCOPY, the procedure call "genhelp.sh X" will create an ETIP help object for each form, menu, or menu item for which there exists a help file in $HELP/X. It will link the object to the help object by way of the help branch, and it will link the help object back to the object by way of the return branch. If there exist in the ETIP prototype any multiple references to a help object, i.e. more than one object whose help branch references the same object, it will remove ALL these redundant references. All of these changes will be made to the X.exp file in $NEWCOPY.

Note: The formatted help files will not be associated with these help objects at this time, that association will be made at compile time. So after genhelp.sh has been applied to a prototype, a subsequent call on ETIP2 will show the existence of the help objects, but will not open them up for inspection since there will be no corresponding text files in the :UT directory.

Attach the formatted help files to the help objects

After doing a save C on a prototype to which we have added the help object H_XXX, we will find there is a generated file called H_XXX.c but no H_XXX.txt, since no formatted file has yet been associated with the help object. However, there is a reference in H_XXX.c to the file "H_XXX.txt". The procedure "modhlplcl.sh X" when run in the directory containing H_XXX.c will rewrite H_XXX.c replacing "H_XXX.txt" with "path/X/H_XXX", where path will be the value of the environmental variable $FMTHELP.

Description of Associated Code

The following procedures are called to interact with the help files.

cleanhelp.sh

cleanhelp.sh modifies a .exp file in NEWCOPY to insure that no help frame is referenced by more than one object

formathelp.sh

formathelp.sh converts the files in $HELP/X to the standard format for help files and inserts them into $FMTHELP/X. It invokes mnufrmdat.sh to determine which of the files in $HELP are actually help files; and it

modifies their names from FX or MX to HX before inserting them into $FMTHELP

genhelp.sh

genhelp.sh modifies a .exp file in NEWCOPY to include help objects for each help file in $HELP/$1 where $1 identifies the prototype

hlpobjlst.sh

hlpobjlst.sh creates a list similar to varlist, but limited to help text objects. The list contains the id, label, and associated file name for each object to which a help branch refers. It takes as input the mnufrmdat file and the .exp file for the application. If the prototype exists in $NEWCOPY, it uses that .exp file; else it uses the .exp file in $CLEAN

linkhelp.sh

linkhelp.sh modifies a .exp file in NEWCOPY to insure that all help frames pass control back to the frame from which they were called, and that they do this when the user presses RETURN.

mnufrmdat.sh

mnufrmdat.sh creates a file to be read by genhelp.sh and genhelp.aw. This file contains the same data as mnufrmlst, i.e. the variable name, id, label, and help id for each form, menu, and menu item in the prototype. However, mnufrmdat is in a form to be read easily by an awk script. Each line, except the last, is a "|" delimited line containing the variable name, object id, and the help branch id of one form, menu, or menu item. The last line contains the next object id to be used for making additions to the .exp file. If the prototype exists in $NEWCOPY, it uses that .exp file; else it uses the .exp file in $CLEAN.

mnufrmlst.sh

mnufrmlst.sh creates a list similar to varlist, but limited to menus and the items associated with them. If the prototype exists in $NEWCOPY, it uses that .exp file; else it uses the .exp file in $CLEAN.

modhlplcl.sh

modhlplcl.sh modifies the .c files for help objects in the local working directory so that they reference the formatted help files in $FMTHELP.

ETIP Database Structure

Many of the procedures for modifying ETIP prototypes depend on a knowledge of the structure of the ETIP database. When you create a prototype using ETIP, all the information you supply is stored in the :UT directory. There are two types of files in this directory: process object scripts, and ETIP database files. The process object scripts contain the scripts for C objects, shell objects, text objects and SQL objects. They are created using an editor like vi, and they can be read and modified using the same editor. The ETIP database files have names like etip.dat1. They cannot be read or modified by use of a text editor, but they can be unloaded into ASCII files using the ETIP import-export tool.

ETIP Import-Export Tool

This tool is supplied by AT&T as an adjunct to ETIP. It consists of the executable files expetip2, impetip2 and impetip2.r all of which are in $ETIPETC along with a documentation file expimp.man. This tool uses the comma as the default field separator in the unloaded files. However, in our awk scripts we use the vertical bar "|" as the field separator, so we have added another executable file, impetip2.bar which generates unloaded files with "|" as the separator.

ETIP Unloaded Database

An unloaded ETIP database is a flat file which contains all the information needed to reconstruct the database. Each line in the file is a "," or "|" separated collection of fields. There are four different types of lines describing respectively the structure of an ETIP object, an item in a menu, a return code branch, or a field in a form. The field structure of each type of line is shown below together with an indication of the function of each field to the extent that we have been able to determine these functions.

Object_line

```
window_01   char(8),
            constant = hwin
object_id_02  char(5),
            format 1:n, where n is unique within this type
```

all references to this object are made using this id

parent_obj_id_03 char(5),

if this object is a submenu, NULL

for all other objects, 0:1

the constant 0:1 seems to imply a primordial menu from which the prototype has been selected

first_obj_04 char(5),

if this object is the first object, then 0:1

for all other objects, NULL

parent_field_id_05 char(5),

if this object is a submenu, then its parent field is entered here

since submenus are attached to fields, they use parent fields not parent objects

object_name_06 char(42),

the string which is displayed in the top border of the object

var_name_07 char(12),

the variable name given to this object in the header

this is the name of the generated .c file, and all references to this object which appear in C programs are based on this name

FIELD_08 integer,

obj_type_09 integer,

70 = FORM

77 = MENU

83 = SHELL

84 = TEXT

285 = SQL_OBJECT

797 = C_OBJECT

FIELD_10 integer,

FIELD_11 integer,

FIELD_12 char(5),

FIELD_13 integer,

FIELD_14 char(16),

save_brnch_15 char(5),

```
        object to which control is passed upon KEY_SAVE or
        KEY_ENTER
cmd_mnu_brnch_16 char(5),
        object to which control is passed upon KEY_COMMAND
help_brnch_17  char(5),
        object to which control is passed upon KEY_HELP
data_string_18  char(62),
        for a c object the name of the first function to be
        executed
prompt_19  char(62),
        the prompt displayed when this object is the active
        object
file_name_20  char(62),
        for a C, SQL, or shell object, the name of the source
        file in :UT
        for a form or menu object, the name of the data file used
        to fill the object
multiple_select_21 integer,
        0=no; 1=yes
number_columns_22 integer,
        number of columns in the menu
slk_label_23  char(10),
slk_label_24  char(10),
slk_label_25  char(10),
slk_label_26  char(10),
slk_label_27  char(10),
slk_label_28  char(10),
        the strings used as labels on the second set of SLKs
FIELD_29  char(20),
FIELD_30  char(20),
slk_brnch_31  char(5),
slk_brnch_32  char(5),
slk_brnch_33  char(5),
slk_brnch_34  char(5),
slk_brnch_35  char(5),
```

```
slk_brnch_36  char(5),
          the objects to which control is passed when one of the
          second set of SLKs is depressed
FIELD_37  char(5),
FIELD_38  char(5),
close_after_use_39 integer,
          89=yes; 78=no
top_left_x_40  integer,
top_left_y_41  integer,
window_width_42  integer,
window_height_43 integer,
          the position and size of the window for this object
FIELD_44  char(20)
```

Menu_Item_Line

```
menu_item_01  char(8),
          constant = hmitrk
object_id_02  char(5),
          format 0:n, where n is unique within this type
          all references to this item are made using this id
parent_menu_id_03 char(5),
          the object to which this menu item is attached
FIELD_04  char(5)
          NULL
item_label_05  char(62),
          the string displayed in the menu for this item
item_branch_06  char(5),
          object to which control is passed upon selecting this
          item
FIELD_07  char(5),
          NULL
item_help_08  char(5),
```

```
          object to which control is passed upon KEY_HELP
FIELD_09  integer,
FIELD_10  integer,
FIELD_11  integer,
FIELD_12  integer,
FIELD_13  char(5),
FIELD_14  integer,
FIELD_15  char(16)
```

Return_Code_Branch_Line

```
rc_branch_01  char(8),
          constant = hmitrk
object_id_02  char(5),
          format 0:n, where n is unique within this type
          all references to this item are made using this id
FIELD_03  char(5),
          NULL
parent_process_id_04 char(5),
          process to which this return code branch is attached
FIELD_05  char(62),
FIELD_06  char(5),
          NULL
return_branch_07 char(5),
          object to which control is passed when the parent process
          returns the value in return_code_09
FIELD_08  char(5),
          NULL
return_code_09  integer,
          return code value which triggers this branch
FIELD_10  integer,
FIELD_11  integer,
FIELD_12  integer,
```

```
FIELD_13  char(5),
FIELD_14  integer,
FIELD_15  char(16)
```

Field Line

```
form_field_01  char(8),
          constant = hfield
object_id_02  char(5),
          format 2:n, where n is unique within this type
          all references to this field are made using this id
parent_id_03  char(5),
          the object to which this field is attached
field_label_04  char(62),
          the string used as a label, caption, for this field
field_prompt_05  char(62),
          the string which appears as prompt when this field is
          active
field_row_06  integer,
          <= 19; the window row in which this field occurs
label_column_07  integer,
          the window column in which the label begins
field_column_08  integer,
          the window column in which the field begins
field_width_09  integer,
          <= 76; the width, i.e. number of characters, of the field
field_type_10  integer,
          the attributes of the field, e.g F_PROTECT, F_NUMERIC,
          etc.
FIELD_12  char(16),
FIELD_13  char(16),
FIELD_14  char(5),
FIELD_15  integer,
```

```
FIELD_16  char(5),
FIELD_17  integer,
FIELD_18  char(16),
data_string_19  char(30)
```

a description of the data set elements used to fill this field

Reuse of Code in ETIP Applications

Efficiency in developing and maintaining applications depends on the ease with which we can reuse code that has already been written and tested. In ETIP this usually means the ability to reuse objects or linked groups of objects since the ETIP object represents the lowest level of functional decomposition.

There seem to be three ways to accomplish this reuse: physically copying the ETIP objects from one part of the application, or a different application, to another; invoking a set of objects which have been created and linked using the ETIP Designer and then stored in a common library; or generating and linking the ETIP objects by an automated process. The first method usually involves some modifications to the copied code at the ETIP Designer level, it is described under Cut and Paste. The second method would be used when we want the linked set of objects to behave the same way every time we call them, but we want to call them from various places within the application, it is described in Reusing ETIP Objects From A Library. The third method is described under DIAL

Cut and Paste

Often when a programmer needs to include a certain function in the code, he finds an implementation of that function in some other program, copies it into the current program, and modifies it as needed. We have written two procedures which can be used to implement such a cut and paste operation in ETIP.

Description of Associated Code

extr.proto.sh

extr.proto.sh extracts from a CLEAN.exp and .src file the description of all objects which can be reached from a specified object, called the base of the extraction. It creates a new set of CLEAN.exp and .src files containing all those objects which have been extracted. These files are created in the current directory.

extr.proto.sh has three parameters. The first is the name of the prototype from which the objects are to be taken. The second is the object name of the specified object. It should be enclosed in double quotes if it contains embedded blanks The third, optional, parameter is the name to be given to the files created by extr.proto.sh,

i.e. if the third parameter is X, then the files X.exp and X.src are created. The default name for these files is EXTR.

merge.proto.sh

merge.proto.sh merges a pair of .exp and .src files into the :UT directory of an ETIP prototype. The first parameter is the name of the .exp and .src files to be merged. These files must be in the current directory. The second parameter is the name of the ETIP prototype, which must be in the current directory. During the merge, merge.proto.sh will prompt the user for new names, both object and variable, for ETIP objects, and for new file names, whenever the old names in the source prototype conflict with names in the target prototype.

Reusing ETIP Objects From A Library

To create a reusable group of ETIP objects we create the objects and link them in a separate prototype using the ETIP editor, do a save C to create the generated .c files, create a C program to act as the entry object for the group, and insert them into the library.

As an example consider the following problem. We want to be able to generate reports from anywhere in the application using the same group of ETIP objects each time. The user is to be presented with the form

```
            Make Report

     Report To Generate: _____
     Print Report (Y/N): _
     Account Number: ______
     Subaccount: _
```

probably as a result of a menu selection.

Valid data for the first item are to be chosen from a submenu which depends on the user's id and the place in the application where the report is being generated. Thus there will be in the $EDIT directory several files that might be used as submenus here.

Print Report (Y/N) determines whether the report is to be

displayed on the screen or sent to a printer.

Account number and Subaccount denotes the account whose data is to be reported. For some users the account number will be fixed and for them this field must be protected.

The first three fields will involve data validation implemented by submenus and the field attributes F_REQUIRE and F_MONLY.

The report will actually be generated by the DBMS report writer and the appropriate report writer command files have already been built in $REPORT.

To prepare a library function with these properties, we created an ETIP prototype called util in which we created three objects: Make Report/F_MR, Make Report Init/C_MRI, and Make Report Execute/C_MRE with the flow of control: C_MRI to F_MR to C_MRE to C_MRI.

As can be seen in the accompanying source code, C_MRI moves the cursor to the top of the form, blanks out all field, and modifies the submenu file name for the first field and the protection attribute for the third field. Notice that it tests the boolean function opened(formrectptr->panel) before invoking update_fields and update_field_flags. If these functions are called when the form is closed, the application will crash.

```
MRI()
{
form_rec *formrecptr;
field_rec *fieldrecptr;

formrecptr = (form_rec *)F_MR.sc_strk;
fieldrecptr = (field_rec *)formrecptr->fields;

/* move cursor to top of form */
set_top_of_form(formrecptr);

/* insert blanks in all fields of the form */
blank_out_form(formrecptr);

/* the make_report choices submenu file defaults to hldrpts.edt, the reports
which are available to holders; if the user is a manager, then the choices
```

```
submenu must be mgrrpts.edt */
if ( has_permission(P_ACCTMGR) ) {
    strcpy ( ms0F_MR.sc_file,"mgrrpts.edt" );
    }
else {
/*  if the user is a holder, then only the curr_accnum can be chosen  */
    strcpy ( fieldrecptr[2].value,curr_accnum );
    fieldrecptr[2].flags |= F_PROTECT;
    }
if ( opened(formrecptr->panel) ) {
    update_fields(formrecptr);
    update_field_flags(formrecptr);
    }
}
```

The source code for C_MRE contains the code needed to invoke the report writer for the DBMS. This C object needs a frame if the report writer is interactive.

```
execute_report_form(argv)
char **argv;
{
   register char *p;
   char command[80];
   char reportpath[120];

   for(p=argv[0];*p && *p != '-';p++);
   for(p--;*p == ' ';p--);
   p++;
   *p = 0;

   if (argv[1][0] == 'N') {
      printf("The report will pause after each screen\n");
      printf("At the colon (:) prompt press RETURN to continue.\n");
      printf("Press RETURN to receive report.\n");
      fflush(stdout);
      getchar();
      /* sprintf(command,"echo \"%s\\012%s\" \| sacego /usr2/isms/pubscntl/
```

```
report/%s | pg",argv[2],argv[3],argv[0]); */
      sprintf(command,"echo \"%s\\012%s\" \| sacego %s/%s |
pg",argv[2],argv[3],getenv("REPORTS"),argv[0]);
      system(command);
   }
   else {
      /* sprintf(command,"echo \"%s\\012%s\" \| sacego /usr2/isms/pubscntl/
report/%s | lp",argv[2],argv[3],argv[0]); */
      sprintf(command,"echo \"%s\\012%s\" \| sacego %s/%s |
lp",argv[2],argv[3],getenv("REPORTS"),argv[0]);
      system(command);
      printf("Report Printed. Press RETURN to continue.\n");
      fflush(stdout);
      getchar();
   }
}
```

Next we issue the ETIP save c command and move the generated C source files for all three objects into the source file for our library. Each of these files contains the line #include "test.h" which ETIP always inserts in the generated C code to enable cross references among the objects in the prototype. However the test.h file which ETIP generates in response to the save c command will not be present in the library source directory. So we must delete the #include lines and include the following declarations.

```
extern screen F_MR;
extern screen C_MRE;
extern screen C_MRI;
extern screen ms0F_MR;
```

The fourth line will not be found in the generated test.h file. It makes visible the submenu for the first field in the form. Submenus are defined as screens but ETIP does not include them in the test.h file. In this case we need an external declaration since C_MRI modifies the file name of the submenu attached to the first field in F_MR. The need for this declaration would show up as a compile time error.

Finally, we need a function to serve as the point of entry for our package. Notice that it opens a window for C_MRI and establishes a link back to the current object, which of course is unknown at the time this code is inserted in the library. When the user depresses the CANCEL key, control will return to the object

from which make_report was invoked.

```
make_report()
{
openwindow(get_cur_obj(),&C_MRI);
}
```

So the complete package to be inserted in the source file for the C functions library looks like this.

```
extern screen F_MR;
extern screen C_MRE;
extern screen C_MRI;
extern screen ms0F_MR;

make_report()
{
openwindow(get_cur_obj(),&C_MRI);
}
```

the complete F_MR.c file exclusive of the include test.h line

the complete C_MRE.c file exclusive of the include test.h line

the complete C_MRI.c file exclusive of the include test.h line

This report generator can be called from anywhere in the prototype by creating one C object as follows

ETIP C object Description

Label: Make Report

Variable name: C_MR

Frame for C: No

Function: make_report()

and linking to that object whenever we want to generate a report.

Database Interactive Application Library (DIAL)

Every ETIP application we have generated so far has required a fair amount of information management. We found that integrating a Database Management System (DBMS) to ETIP can be tedious, repetitive and error prone. In addition the speed of the built-in DBMS left quite a bit to be desired.

We found the most efficient way of incorporating DBMS functions into an ETIP application was to use the Embedded SQL (ESQL) package contained with the Informix DBMS. ESQL allowed us to write SQL statements and manipulate database information with reasonable speed and accuracy.

Using ESQL unfortunately precluded us from using any of the built-in features of ETIP, such as data strings for filling form fields. So we started making calls directly to the ETIP library to fill in ETIP fields and edit flags.

As we progressed we found that in a typical application we repeated the same type if ESQL code over and over again for displaying and updating database information from ETIP applications. This repetition increased the final application code size, and often introduced errors in the repeated and copied code.

As a result we finally developed the Database Interface Applications Library (DIAL). DIAL has the following functions:

1. DIAL allows an ETIP applications programmer to quickly specify the interface characteristics between a database and an ETIP form.

2. The library takes that specification and at execution time allows the user to specify selection criteria for a portion of the database.

3. The library retrieves the selected database information and transfers it to an ETIP form which is then displayed in a multipage format.

4. The user can modify the data on the ETIP form and request that the updated information be saved to the database.

5. The library will save any updated information into the database and leave the ETIP form.

As a result of incorporating these features into DIAL we have found the following positive effects:

1. Since the applications programmer generates a minimum of code many of the errors that are generated from repetitive code are eliminated.

2. Because the library in included in the application only once a great decrease in code size is often realized.

3. All of the ESQL code necessary to perform the library's function is incorporated into the library. As a result the applications programmer has no need to learn ESQL resulting in a faster turn around time for ETIP applications.

Library Functions

ETIP includes several libraries and for two of them, libEc.a and libEt.a, it provides the source code and a Makefile. We have added a number of functions to these libraries, mostly for the purpose of controlling the action of ETIP forms. We describe below the functions we have added as well as some of the functions in the ETIP distribution libraries.

ETIP Form Field Population Guidelines

One of the basic operations in ETIP applications is form field population. However the ETIP designer provides only a single option, data strings, for populating fields with information from the data base or C objects. However, in general, it's awkward and unmanageable.

A much better method of populating form fields is through the ETIP library's set of C routines. These provide a relatively painless and more easily manageable way to populate form fields.

The ETIP header file Et.h must be included in any C object used to populate forms. Two structure definitions of interest are found in the header file, the form_rec and the field_rec. These structure definitions (shown below) are used by the ETIP library functions to populate form fields.

Form and field structure definitions

```
typedef struct {
char            *name;          /* field name                      */
int             row;            /* field row                       */
int             ncol;           /* name column                     */
int             fcol;           /* field column                    */
int             len;            /* field length                    */
int             flags;          /* field flags                     */
char            *value;         /* field value (initial/return)    */
menu_rec        *menu;          /* optional menu pointer           */
char            *prompt;        /* field prompt                    */
char            *rptr;          /* reserved pointer                */
char            *uptr;          /* user pointer                    */
funcptr         init;           /* initialization call-back        */
funcptr         term;           /* termination call-back           */
char            *str;           /* reserved (for reg. exp., etc)   */
FIELDTYPE       **ftype;        /* ETI field type                  */
int             maxlen;         /* max length for scroll-field     */
} field_rec;

typedef struct {
```

```
char            *label;         /* form label                      */
int             flags;          /* form flags                      */
int             y;              /* desired window y-coordinate     */
int             x;              /* desired window x-coordinate     */
field_rec       *fields;        /* fields                          */
field_rec       *dfield;        /* current field                   */
slk_rec         *slks;          /* pointer to desired slks         */
FORM            *form;          /* pointer to ETI FORM             */
PANEL           *panel;         /* pointer to ETI PANEL            */
char            *rptr;          /* reserved pointer                */
char            *uptr;          /* user pointer                    */
} form_rec;
```

Each form_rec has an array of field_rec structures which describe each field in the form. C objects can change the value, flags, and str fields of the field_rec structures.

In the generated C code both the form structure and the field array are given names based on the C variable name of the form.

The C variable name of the form structure is the C variable name preceded by a lowercase z (i.e. if the C variable name is F_myform then the form structure name is zF_myform).

Similarly the name of the field array is the C variable name preceded by a lowercase f (i.e. fF_myform in the previous example).

In addition a pointer to the field array is always stored in the form structure in field "fields", so fF_myform and zF_myform.fields have the same value and can be used interchangeably.

Be aware that labels in the form, such as titles, are considered to be fields with a field width of 0. These label-only fields must be accounted for when populating forms.

Creating Custom Screen Labeled Keys (SLKS) in ETIP

ETIP has default SLKS for menu, form, and text objects. Unfortunately many of the functions available on the default SLKS are not necessary for most screens. Also in many screens a useful feature could be encoded on an SLK. This file describes how to create and maintain custom SLKS.

The default SLKS are defined in the ETIP library. Each SLK is represented by a label and an associated key. There are two arrays defining the 8 available SLKS for each menu, form, and text object. The C variable names are as follows:

Label	KeyFile

Menu (No mark)	m1sys_slks	m1sys_reqs	t_subs.c
Menu (with mark)	m2sys_slks	m2sys_reqs	t_subs.c
Forms	fsys_slks	fsys_reqs	t_subs2.c
Text	tsys_slks	tsys_reqs	t_subs3.c

In the default ETIP library, these arrays are statically defined. To make Custom SLK sets it's necessary to make the above variables mobile. To do this simply define them as C pointers. Then these variables can be pointed at different SLK definitions at run time. See the PUBSS ETIP library for an example.

Managing a custom SLK by hand is tedious work because of the need to restore a custom set of SLKS each time control is returned to a screen object. A couple of functions can help facilitate managing custom SLKS. The basic idea is to maintain a stack of screens and the SLKS associated with them. Each time a new screen is displayed, the SLKS information about the that screen is pushed on a stack. When the current screen is removed the SLKS from the screen underneath are restored from the information about it in the stack. In PUBSS these functions are implemented in the object "DA12;Master Control;Clear SLKS;C". They are currently implemented only for forms but the idea is extensible to menu and text objects.

Description of Associated Code

The following procedures are called to handle form manipulation.

blank_out_form

blank_out_form(form_rec *) can be invoked within any object. It will replace each field in the form with a string containing one blank character This differs from the ETIP supplied function clr_form which replaces each field by a null string. The difference between these is apparent when you have both F_MONLY and F_REQUIRE on a field. If the field contains null then it will be populated by the first item in the choices menu file. If it contains a blank string, it will be left blank but an error message will appear indicating invalid data and the availability of choices.

lock_cursor

lock_cursor(form_rec *; field_rec *) will lock the cursor on the field identified by its arguments. The cursor will remain locked on that field until unlocked by unlock_cursor (see below). Multiple locking of a

screen has no adverse effects because all lock calls after the first are ignored.

THE FORM THAT IS LOCKED MUST BE VISIBLE ON THE SCREEN. LOCKING NONVISIBLE FORMS WILL CAUSE THE APPLICATION TO CRASH!

N.B. Note that pointers are used for both the form_ptr and field_ptr parameters. Do not pass a structure to lock_cursor.

unlock_cursor

unlock_cursor(form_rec *) restores the screen to its original state. Multiple unlocks have no effect upon the screen.

THE FORM THAT IS TO BE UNLOCKED MUST BE VISIBLE ON THE SCREEN. UNLOCKING NON VISIBLE FORMS WILL CAUSE THE APPLICATION TO CRASH!

N.B. Note that pointers are used for the form_ptr parameters. Do not pass a structure to unlock_cursor.

lock_cursor_set

lock_cursor_set(form_rec *; int *) will lock the cursor within the field set whose array is pointed to by unlock_set. The cursor will remain locked within that field set until unlocked by unlock_cursor_set (see below). Multiple locking of a screen has no adverse effects because all lock calls after the first are ignored.

The lock set is specified by an integer array that contain the indexes of the fields that are not to be locked. So if the 5th and 6th field of a form are not to be locked then 5 and 6 must be in the lock_set array. The array is terminated by a negative integer. An integer array was used in order to facilitate initialization at compile time. Pointers cannot be initialized at compile time.

N.B. The current field of the form when lock_cursor_set is called cannot be locked. It will remain unlocked even if it isn't included in the unlock set.

THE FORM THAT IS LOCKED MUST BE VISIBLE ON THE SCREEN. LOCKING NON VISIBLE FORMS WILL CAUSE THE APPLICATION TO CRASH!

N.B. Note that pointers are used for both the form_ptr

and field_ptr parameters. Do not pass a structure to lock_cursor.

unlock_cursor_set

unlock_cursor_set (form_rec *) restores the screen to its original state. Multiple unlocks have no effect upon the screen.

THE FORM THAT IS TO BE UNLOCKED MUST BE VISIBLE ON THE SCREEN. UNLOCKING NON VISIBLE FORMS WILL CAUSE THE APPLICATION TO CRASH!

N.B. Note that pointers are used for the form_ptr parameters. Do not pass a structure to unlock_cursor.

fill_field

fill_field (field_rec *; char *) will populate the given field value with the given string. It makes a copy of str. It can be used at any time in an application. If the form is not on the screen then the values filled in by this function will appear the next time the form is opened. However if the form is currently visible it is necessary to call update_fields (see below) to change the values on the screen to match the filled values.

N.B. A pointer to the field is passed as the first argument, not the field structure itself!

update_fields

update_fields (form_rec *) copies the field data from the internal ETIP form data structure to the on-screen form.

THIS FUNCTION CAN ONLY BE USED WHEN THE FORM IS CURRENTLY ON THE SCREEN! IT WILL CAUSE THE APPLICATION TO CRASH IF THE FORM IS NOT VISIBLE!

N.B. A pointer to the form is passed as the first argument, not the form structure itself!

update_field_flags

update_field_flags (form_rec *) copies the field flags data from the internal ETIP form data structure to the on-screen form. Field flag data can be manipulated directly in the field structures. But like the field values they won't take effect on the screen if it's currently visible.

THIS FUNCTION CAN ONLY BE USED WHEN THE FORM IS CURRENTLY ON THE SCREEN! IT WILL CAUSE THE APPLICATION TO CRASH IF THE FORM IS NOT VISIBLE!

N.B. A pointer to the form is passed as the first argument, not the form structure itself!

master_control_push_slks

master_control_push_slks (slk_rec *; int *; screen *) pushes the current default label, key, and screen information onto the stack and sets the default label and key pointers to the parameters. When the screen is subsequently displayed the labels and keys defined will be used for the SLKS

master_control_clear_slks

master_control_clear_slks() removes the screen on the top of the stack from view and restores the default label and key pointers to the values on the top of the stack. The stack is popped. This action restores the label and key definitions from the previous screen. Since this function has no parameters, it can be called as a C object. The Cancel function of screen with custom SLKS calls clear_slks.

Appendix - Environmental Variables

Environmental variables, unix variables that allow string substitutions, were used to navigate through the areas as we set up the working environment for the applications and configuration control.

The following environmental variables are used in our procedures.

ETIPDIR - the root of the ETIP libraries, /usr/ACE/ETIP

ETIPETC - the directory which contains most of the scripts and programs used to manage applications

ETIPLIB - the libraries used in compiling and linking

ETIPSRC - the source code for ETIPLIB

EtipInc - the include code for both ETIPLIB and the application specific code

CLEAN - this directory must be present in each application. It contains a complete but transportable, i.e. ASCII, copy of each prototype in the application. It also contains several previous generations of each prototype.

NEWPROTO - the newproto directory is the working area where modifications are made to prototypes. It contains a sub-directory for each prototype in the application.

LOCK - the lock area is a directory which contains a file for each prototype currently in the newproto area. These lock files prevent simultaneous attempts to modify a prototype.

NEWCOPY - the newcopy directory contains the latest version of a prototype after the programmer has completed making modifications to it but before the application administrator has restored it to the clean area.

EDIT - the edit area contains the master files for all choices submenus in the application.

HELP - the help area contains the source files for all help text objects in the application.

FMTHELP - the format help area contains the data to be displayed in all help objects. Each file contains the data from one file in $HELP but modified to include paging and prompting information.

FORMS - the forms area contains the files needed to define all the fields in the forms of the applications.